Estruturas de Dados e Algoritmos

Capítulo 6

Árvores Binárias de Pesquisa

5. Árvores Binárias de Pesquisa

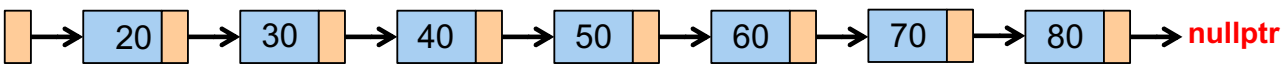
- Qual o interesse das estruturas em árvore?
- Árvores – conceitos gerais
- Árvores binárias
- Programação de uma classe de objetos para manipular árvores binárias de pesquisa
- Exercícios e tópicos adicionais

5. Árvores Binárias de Pesquisa

- Qual o interesse das estruturas em árvore?

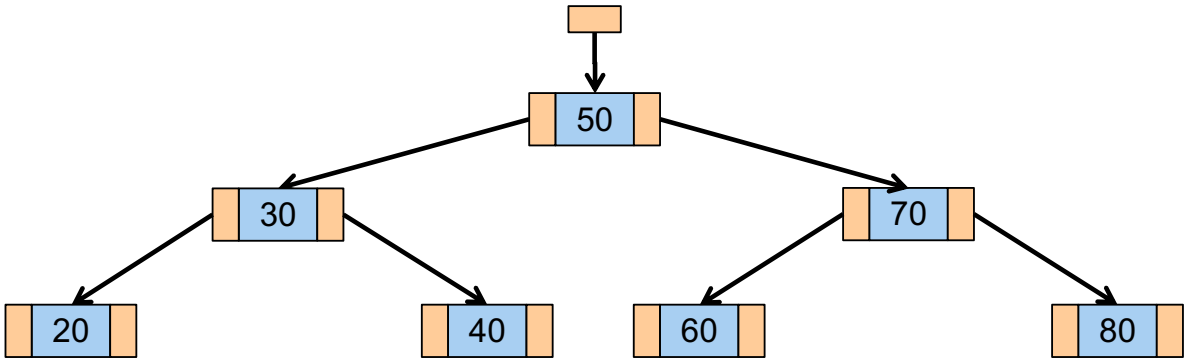
Limitação das Listas Ligadas

- As listas ligadas têm a grande vantagem de permitirem a alocação dinâmica de memória, mas apresentam uma limitação importante...
  - Acesso sequencial aos dados:
    A pesquisa numa lista ligada ordenada é feita percorrendo a lista desde o início até se encontrar o elemento pretendido...



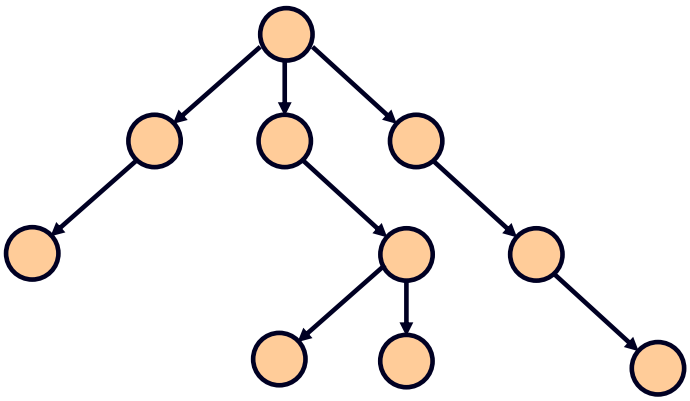
Árvores

- Estruturas de dados dinâmicas e ligadas, com organização hierárquica, que permitem otimizar a pesquisa de dados
- Exemplo de uma árvore que permite realizar facilmente pesquisa binária:



Árvores

- São frequentemente utilizadas para organizar informação hierarquicamente
- Exemplos: árvore genealógica, organigrama de uma organização de pessoas, *router-tables*, etc.

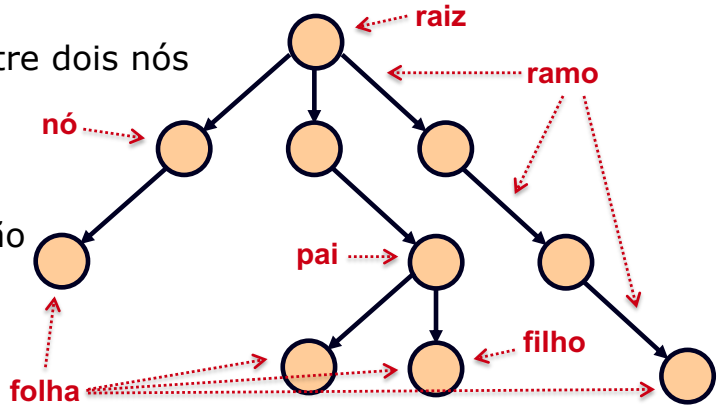


5. Árvores Binárias de Pesquisa

- Qual o interesse das estruturas em árvore?
- Árvores – conceitos gerais

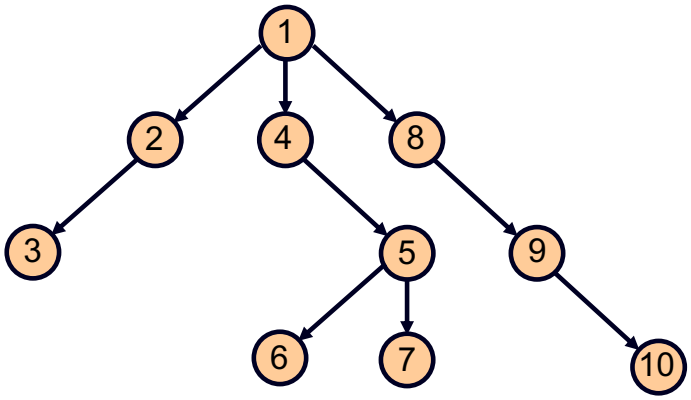
Árvores

- Terminologia:
  - **Nó**: cada elemento da árvore é representado por um nó
  - **Raiz**: primeiro nó da árvore; não recebe nenhuma ligação
  - **Ramo**: ligação orientada entre dois nós
  - **Pai**: origem de uma ligação
  - **Filho**: destino de uma ligação
  - **Folha**: nó sem filhos



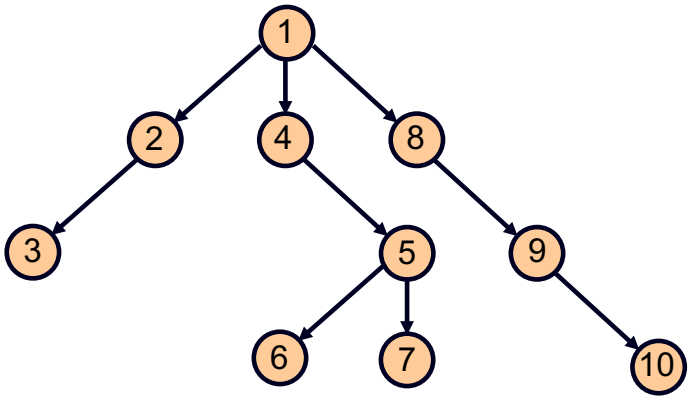
Árvores

- Terminologia:
  - **Caminho**: sequência de ligações entre pares de nós, ex. 1 → 4 → 5 → 7
  - **Comprimento de um caminho**: número de ligações do caminho
  - **Profundidade de um nó**: comprimento do caminho entre a raiz e esse nó. Por definição, a profundidade da raiz é igual a zero.



Árvores

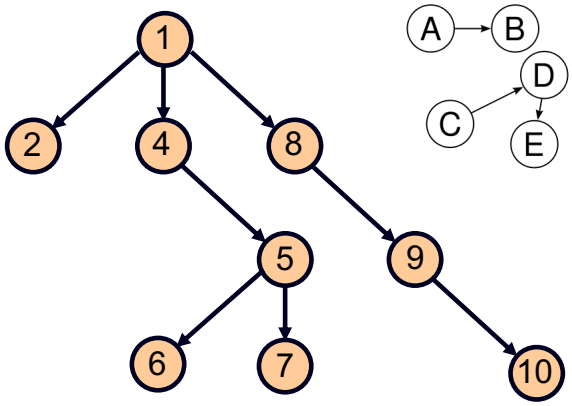
- Terminologia:
  - **Profundidade de uma árvore**: é igual à profundidade da sua folha mais profunda.
  - **Altura de um nó**: comprimento do maior caminho entre esse nó e todas as folhas suas descendentes
  - **Altura de uma árvore**: é igual à altura da raiz. É sempre igual à profundidade da árvore.



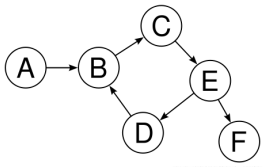
Árvore – Estrutura de Dados

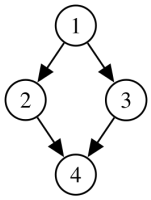
- Definição recursiva:

  Coleção de nós (a começar pela raiz) em que cada nó é uma estrutura de dados que consiste num valor e uma lista de referências para nós (os filhos) com a restrição de nenhuma referência ser duplicada e de nenhuma destas referências apontar para a raiz



Por terem ciclos, não são árvores:



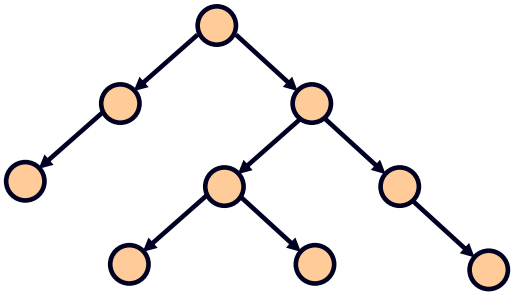


5. Árvores Binárias de Pesquisa

- Qual o interesse das estruturas em árvore?
- Árvores – conceitos gerais
- Árvores binárias

Árvore Binária

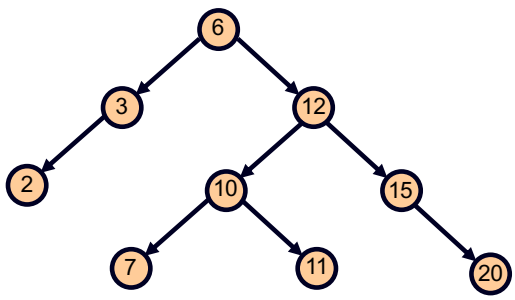
- **Definição**: Uma árvore em que cada nó tem no máximo dois filhos



- Se os dados armazenados numa árvore binária obedecerem a um critério de ordenação, então diz-se que essa árvore é uma **árvore binária de pesquisa** (do Inglês, *binary search tree* – BST).

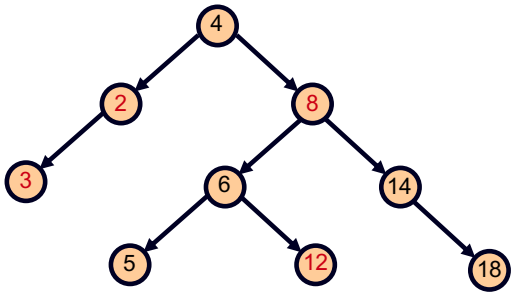
Árvore Binária

- Numa **árvore binária de pesquisa** as seguintes proposições são *sempre* verdadeiras para *qualquer* um dos seus nós:
  1. O valor armazenado no nó é maior (menor) do que todos os valores armazenados na subárvore esquerda
  2. O valor armazenado no nó é menor (maior) que todos os valores armazenados na subárvore direita
  3. Ambas as subárvores da direita e da esquerda são também elas próprias árvores binárias de pesquisa

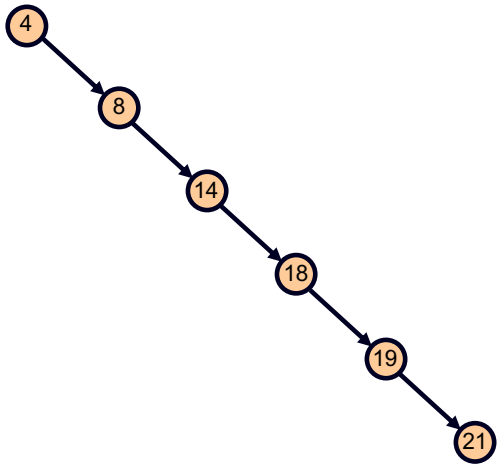


Árvore Binária

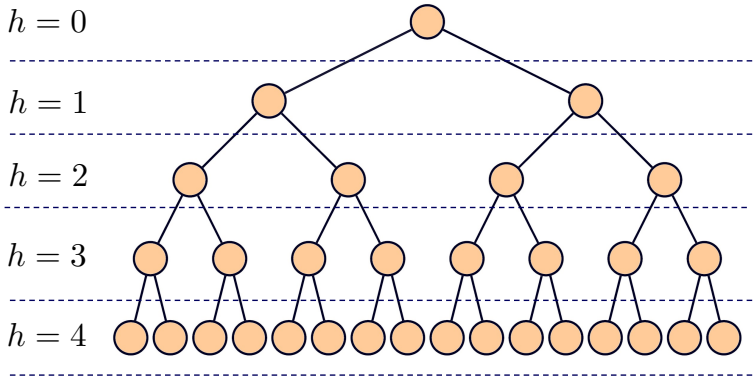
- Esta árvore *não é* uma árvore binária de pesquisa:



- Caso especial de uma árvore binária de pesquisa:



Altura vs. Número de Nós da Árvore



Altura	Número de nós
0	$n_0 = 2^0 = 1$
1	$n_1 = 2^1 + n_0$
2	$n_2 = 2^2 + n_1$
3	$n_3 = 2^3 + n_2$
...	...
h	$n = 2^h + n_{h-1}, \forall\ h \in \mathbb{N}$

$$\vdots$$

$$n = 2^{h+1} - 1 = \sum_{i=0}^{h} 2^i$$

$$h = \log_2(n+1) - 1 \simeq \log_2 n$$

- Pesquisa tem complexidade *O(log n)* p/ árvore; *O(n)* p/ lista ligada
- Exemplo:
  - *h = 20, n = 2097151, h/n ≃ 10^{-5} !*

Árvore Binária de Pesquisa

- Principais operações definidas sobre uma árvore binária de pesquisa:
  1. *Pesquisa* de um item
  2. *Travessia* da árvore por uma determinada ordem
  3. *Inserção* de um novo nó
  4. *Eliminação* de um nó
- Utiliza-se muitas vezes algoritmos recursivos para implementar estas operações

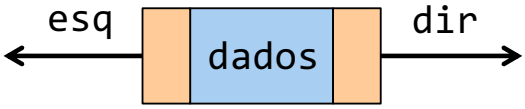
5. Árvores Binárias de Pesquisa

- Qual o interesse das estruturas em árvore?
- Árvores – conceitos gerais
- Árvores binárias
- Programação de uma classe de objetos para manipular árvores binárias de pesquisa

Árvore Binária de Pesquisa

- Definição da classe `CNoArvoreBinaria`

```
template <class T>
class CNoArvoreBinaria {
 public:
   T dados; // espaço para armazenar os dados
   CNoArvoreBinaria<T> *esq; // ponteiro p/ filho da esquerda
   CNoArvoreBinaria<T> *dir; // ponteiro p/ filho da direita
};
```



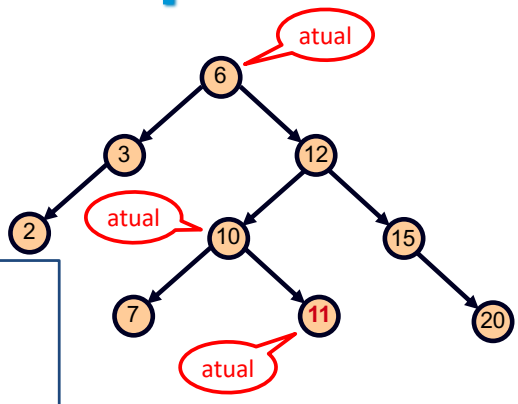
Árvore Binária de Pesquisa

- Definição da classe `CArvoreBinaria`

```
template <class T> class CArvoreBinaria {
   CNoArvoreBinaria<T> *raiz; // ponteiro para a raiz
   bool procuraRecursiva(CNoArvoreBinaria<T>*, const T&) const;
   void escrevePorOrdem(CNoArvoreBinaria<T>*) const;
   void escrevePreOrdem(CNoArvoreBinaria<T>*) const;
   void escrevePosOrdem(CNoArvoreBinaria<T>*) const;
   void insere(const T&, CNoArvoreBinaria<T>*);
   void destroiSubArvore(CNoArvoreBinaria<T>*);
 public:
   CArvoreBinaria();
   ~CArvoreBinaria();
   bool procura(const T&) const;
   void escrevePorOrdem(void) const;          // public front-end
   void escrevePreOrdem(void) const;          // public front-end
   void escrevePosOrdem(void) const;          // public front-end
   void insere(const T&);                     // public front-end
   bool elimina(const T&);
   void destroi(void);
};
```

Árvore Binária de Pesquisa

- Pesquisa de um item: *versão iterativa*



1. atual ← raiz
2. **Enquanto** atual <> nullptr
3. **Se** atual->dados == item **Então** devolve **verdadeiro**
4. **Senão Se** atual->dados<item **Então** atual ← atual->dir
5. **Senão** atual ← atual->esq;
6. **Fim Se**
7. **Fim Enquanto**
8. Devolve **falso**

Árvore Binária de Pesquisa

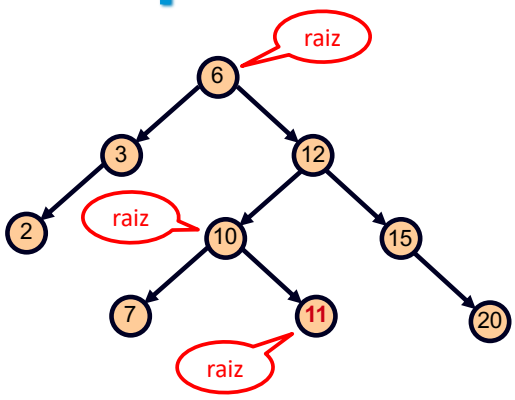
- Pesquisa de um item: *versão iterativa*

```
template <class T>
bool CArvoreBinaria<T>::procura(const T& item) const {
   CNoArvoreBinaria<T> *atual;

   atual = raiz;
   while (atual != nullptr) {
      if (atual->dados == item) return true;  // encontrou
      // vai procurar numa das subárvores, direita ou esquerda
      if (atual->dados < item) atual = atual->dir;
      else atual = atual->esq;
   }
   return false; // não encontrou
}
```

Árvore Binária de Pesquisa

- Pesquisa de um item: *versão recursiva*



1. **Se** raiz == nullptr **Então** devolve **falso**
2. **Fim Se**
3. **Se** raiz->dados == item **Então** devolve **verdadeiro**
4. **Fim Se**
5. **Se** raiz->dados < item
   Então devolve *procura na subárvore direita*
6. **Senão** devolve *procura na subárvore esquerda*
7. **Fim Se**

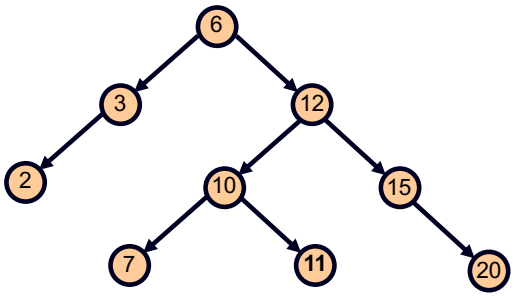
Árvore Binária de Pesquisa

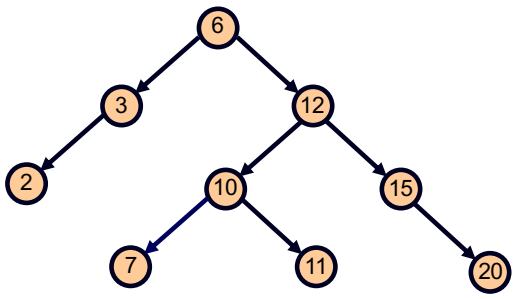
- Pesquisa de um item: *versão recursiva*

```
template <class T>
bool CArvoreBinaria<T>::procuraRecursiva(
        CNoArvoreBinaria<T> *raiz, const T& item) const {
   // casos elementares
   if (raiz == nullptr) return false;
   if (raiz->dados == item) return true;
   // caso geral (pesquisa na subárvore direita ou esquerda)
   if (raiz->dados < item)
        return procuraRecursiva(raiz->dir, item);
   else return procuraRecursiva(raiz->esq, item);
}
```

Árvore Binária de Pesquisa

- Travessia da árvore por-ordem (em-ordem, *in-order*, E-R-D)





2, 3, 6, 7, 10, 11, 12, 15, 20

1. **Se** raiz <> nullptr **Então**
2. escreve por-ordem a subárvore esquerda
3. escreve raiz->dados
4. escreve por-ordem a subárvore direita
5. **Fim Se**

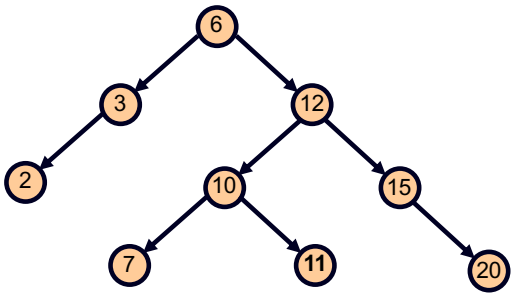
Árvore Binária de Pesquisa

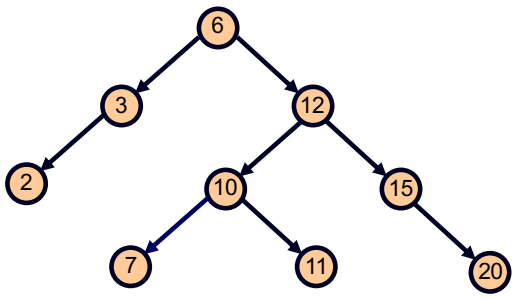
- Travessia da árvore por-ordem

```
template <class T>
void CArvoreBinaria<T>::escrevePorOrdem() const {
   escrevePorOrdem(raiz);
}
//-------------------------------------------------
template <class T>
void CArvoreBinaria<T>::escrevePorOrdem(
                           CNoArvoreBinaria<T>* raiz) const {
   if (raiz == nullptr) return;  // caso elementar
   escrevePorOrdem(raiz->esq);   // senão caso geral...
   cout << raiz->dados << endl;
   escrevePorOrdem(raiz->dir);
}
```

Árvore Binária de Pesquisa

- Travessia da árvore pré-ordem (*pre-order*, R-E-D)





6, 3, 2, 12, 10, 7, 11, 15, 20

1. **Se** raiz <> nullptr **Então**
2. escreve raiz->dados
3. escreve pré-ordem a subárvore esquerda
4. escreve pré-ordem a subárvore direita
5. **Fim Se**

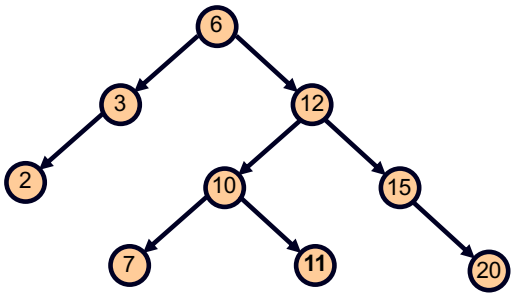
Árvore Binária de Pesquisa

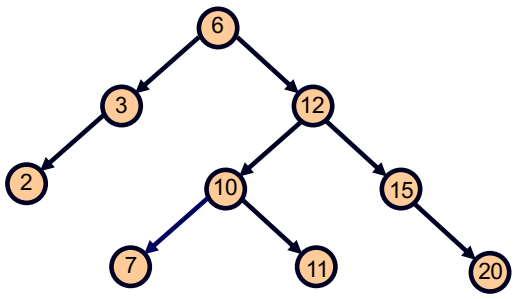
- Travessia da árvore pré-ordem

```
template <class T>
void CArvoreBinaria<T>::escrevePreOrdem() const {
   escrevePreOrdem(raiz);
}
//-------------------------------------------------
template <class T>
void CArvoreBinaria<T>::escrevePreOrdem(
                          CNoArvoreBinaria<T>* raiz) const {
   if (raiz == nullptr) return;     // caso elementar
   cout << raiz->dados << endl;     // senão caso geral...
   escrevePreOrdem(raiz->esq);
   escrevePreOrdem(raiz->dir);
}
```

Árvore Binária de Pesquisa

- Travessia da árvore pós-ordem (*post-order*, E-D-R)





2, 3, 7, 11, 10, 20, 15, 12, 6

1. **Se** raiz <> nullptr **Então**
2. escreve pós-ordem a subárvore esquerda
3. escreve pós-ordem a subárvore direita
4. escreve raiz->dados
5. **Fim Se**

Árvore Binária de Pesquisa

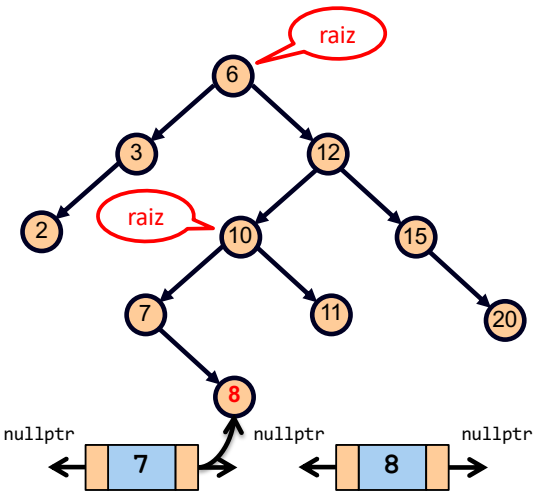
- Travessia da árvore pós-ordem

```
template <class T>
void CArvoreBinaria<T>::escrevePosOrdem() const {
  escrevePosOrdem(raiz);
}
//-------------------------------------------------
template <class T>
void CArvoreBinaria<T>::escrevePosOrdem(
                          CNoArvoreBinaria<T>* raiz) const {
  if (raiz == nullptr) return;    // caso elementar
  escrevePosOrdem(raiz->esq);     // senão caso geral...
  escrevePosOrdem(raiz->dir);
  cout << raiz->dados << endl;
}
```

Árvore Binária de Pesquisa

- Inserção de um item numa árvore não vazia

Ex. Inserir 8:

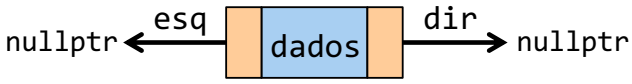



```
1.  Se item < raiz->dados Então
2.      Se raiz->esq <> nullptr Então insere(raiz->esq, item)
3.      Senão
4.          Cria um novo nó sem filhos e novo->dados ← item
5.          raiz->esq = novo
6.      Fim Se
7.  Senão Se item > raiz->dados Então
8.      Se raiz->dir <> nullptr Então insere(raiz->dir, item)
9.      Senão
10.         Cria um novo nó sem filhos e novo->dados ← item
11.         raiz->dir = novo
12.     Fim Se
13. Fim Se
```


Árvore Binária de Pesquisa

- Inserção de um item numa árvore (método `public`)

```
template <class T>
void CArvoreBinaria<T>::insere(const T& item) {
   if (raiz != nullptr) insere(item, raiz); // faz o trabalho...
   else {  // cria o primeiro nó da arvore (a raiz)
      raiz = new CNoArvoreBinaria<T>;
      raiz->dados = item;
      raiz->esq = raiz->dir = nullptr;
   }
}
```



Árvore Binária de Pesquisa

- Inserção numa árvore não vazia (método `private`)

```
template <class T> void CArvoreBinaria<T>::insere(
                      const T& item, CNoArvoreBinaria<T> *raiz) {
   if (item < raiz->dados)  // raiz tem de ter dados!...
      if (raiz->esq != nullptr) insere(item, raiz->esq);
      else { // novo nó é o primeiro nó da subárvore esquerda
         raiz->esq = new CNoArvoreBinaria<T>;
         raiz->esq->dados = item;
         raiz->esq->esq = raiz->esq->dir = nullptr;
      }
   else if (item > raiz->dados)
      if (raiz->dir != nullptr) insere(item, raiz->dir);
      else { // novo nó é o primeiro nó da subárvore direita
         raiz->dir = new CNoArvoreBinaria<T>;
         raiz->dir->dados = item;
         raiz->dir->esq = raiz->dir->dir = nullptr;
      }
}
```